Design and implementation of a simple mesh Network-on-Chip

Project report for 02211 Advanced Computer Architecture F12

Graeme Best, Matt Birman, Oscar Rahnama and Wojciech Pawlak
Department of Informatics and Mathematical Modelling
Technical University of Denmark
Asmussens Alle, Building 305, DK-2800 Lyngby, Denmark
{s114581, s114580, s114516, s091820}@student.dtu.dk

***Abstract* - We present the design and implementation of a 16-node mesh-topology Network-on-Chip. The focus is on a simple design and therefore a reduction in the number of logic units. An interface protocol has been designed to allow the implementation of different computer systems. An example network and testing results are presented.**

Index Terms – Network-on-Chip, mesh topology, fixed priority routing, deterministic routing

1. Introduction

The goal of this project work in course 02211 Advanced Computer Architecture has been to design and implement a functioning Network-on-Chip (NoC) running on an FPGA. The project has been implemented on an Altera DE2-70 development board using an Altera Cyclone II FPGA integrated circuit with a 50 MHz clock. Programming has been done in the Altera Quartus design software using the VHDL hardware description language. With minor adjustments it could be implemented on other FPGA boards.

The project has been made available as open-source, and can be accessed from:
github.com/mattbirman/Network-on-Chip-in-VHDL

Our main aim from the beginning was to keep the design as simple as possible to reduce the implementation complexity and therefore reduce the number of required logic units. For this reason we chose to use:

- simple mesh topology
- fixed priorities in the routing algorithm,
- single-flit packet size,
- fixed number of nodes,
- fixed address and data length in packets
- evenly divided address space

Some network arrangements have been tried using example IP cores (for example traffic generators, memory modules and UART modules) and a real-time MATLAB program has been created to visualise the network traffic during testing.

2. Architecture

A Network-on-Chip can be seen as a distributed system implemented and embedded on a chip. It is a Multi-Processor System-on-Chip (MPSoC) with cores communicating through an interconnect fabric. The system consists of a network of nodes that need to send and receive data in order to complete their actions. These systems have a variety of requirements since the devices in nodes can be of different types, such as processors, memory or input/output devices. Packet switching is used as a communication method across the network, with packets used as the communication medium. Thus the latency or throughput of communication has to be ensured for the system to be usable.

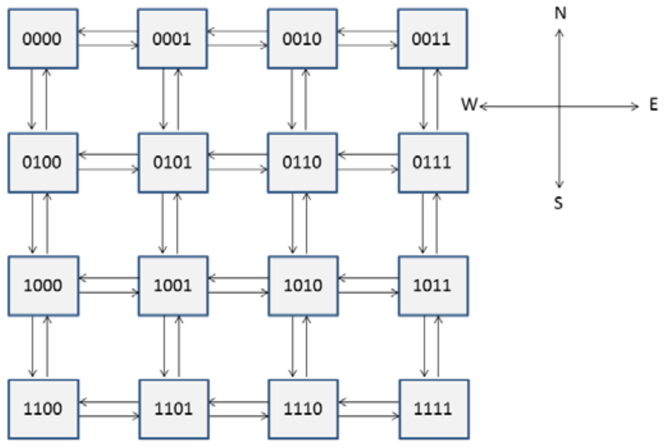
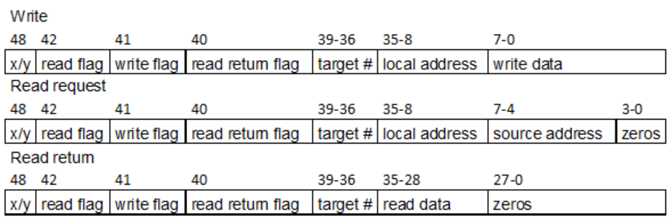


Fig. 1 General perspective on 16-nodes Network-on-Chip

The implemented network is divided into 16 nodes (see Fig.1) and the nodes consist of an IP core, a network adapter and a router. A network adapter is used to connect a device to the router and a router connects the node to the rest of the network.

2.1. Packet format

There are three types of packets in the network: write, read request and read return, as shown in Fig.2. The write packet is sent by a *master* to write data to a *slave*. A read request is sent by a *master* to a *slave* and a *slave* then replies with a read return packet. All packets have a fixed length of 49 bits long but some formats require padding with zeros.



Write

48	42	41	40	39-36	35-8	7-0
x/y	read flag	write flag	read return flag	target #	local address	write data

Read request

48	42	41	40	39-36	35-8	7-4	3-0
x/y	read flag	write flag	read return flag	target #	local address	source address	zeros

Read return

48	42	41	40	39-36	35-28	27-0
x/y	read flag	write flag	read return flag	target #	read data	zeros

Fig. 2 Packet formats

All packets have three flags (one for each type) that indicate what type of packet they are. No more than one of the three flags should be a ‘1’ and an empty packet is indicated when they are all set to ‘0’.

Write and read request packets require a destination address. The system has a 32 bit address space where the first 4 bits are used as the unique node IDs. The remaining 28 bits are used for local addresses. The read request packet also contains the source ID so the receiver knows where to send the read return. Read return packets only contain the destination node ID of the source it is replying to.

Packets also have 6-bits for the routing counters: 3-bits each for the x and for the y directions of the grid. These counters are initialised by the network adapters and decremented by the routers. The network has a fixed size of 4x4 nodes, so 2 bits are needed to encode the position in the grid and 1 bit to indicate the direction for each axis. If a larger mesh were implemented, the number of bits would have to be increased.

A write packet contains 8-bits of write data while a read return contains 8-bits of read data.

The packets are sent over one clock cycle rather than broken up over multiple clock cycles, to keep the hardware and logic simple. Therefore packets do not have to contain any additional information to help the routers and network adapters to decode a stream of packet flits.

2.2. Node

A node is one logical point in the network. It contains a device, such as a memory block or a processor, as well as a network adapter and a router. The details of these components are described in the following sections.

2.3. Network Adapters

The network adapters are the interface between the IP core and the router and their purpose is to convert signals from the local bus into a packet format suitable for the network and back again. There are two types of network adapters; ***master*** and ***slave***.

2.3.1 Master

The master network adapter (see Fig. 3) receives the following signals from the master device: *write address*, *write enable*, *write data* and *read request*. A master device can connect to the network adapter and the network should be totally transparent. The NA sends back *not ready*, *read return* and *read data*. Any device wishing to connect to the network needs to handle a *not ready* signal from the network adapter. Its output interface with a router is a *packet out*. It can receive a busy signal from the router and a read return packet.

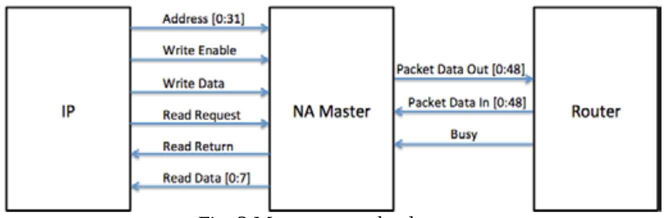


Fig. 3 Master network adapter

2.3.2. Slave

The slave network adapter (see Fig. 4) communicates with the slave device through the use of *address*, *write enable*, *write data* and *read request* output signals. It receives *slave not ready*, *read return* and *read data* from the slave device. As for outputs signals to the router, the slave network adapter sends *packet data out* and *network adapter not ready*. It receives *packet data in* and *router not ready* as inputs from the router.

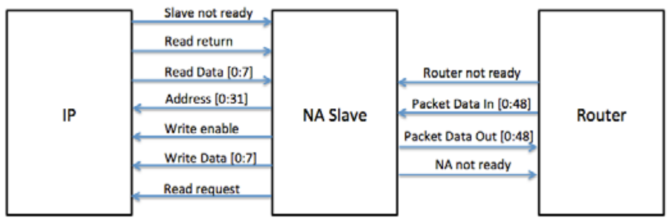


Fig. 4 Slave network adapter

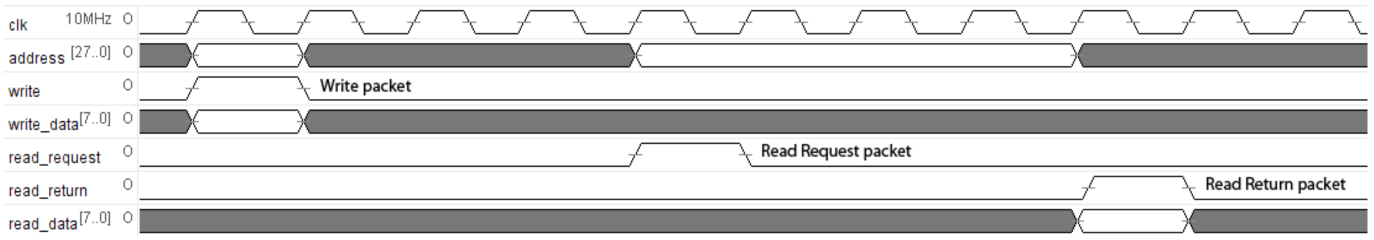


Fig. 5 Timing diagram for master network adapter

2.3.3 Route Calculation

The network adapter is also responsible for determining the route of the packet that it wants to send. Every node on the network has a unique node ID. The network adapter determines the values that it must assign to the *x* and *y* counters of the packet based on its own node ID and the packet destination node ID. To obtain the value of the *y* counter, the network adapter looks at the first two bits of the destination node ID of the packet and then subtracts it from the first two bits of its own node ID. If the resulting value is negative, it is placed in the y-counter section of the packet and preceded with a 1. If the value is positive, the resulting binary value is preceded with a 0. A similar process is used when calculating the value for the x-counter of the packet. The only difference is that we use the second pair of two bits of the node IDs for the calculations as they refer to the x-position of the node on the grid.

2.3.4 Interface

To connect a new IP core to the network, it needs to be able to connect to the interface defined by the network adapters. Alternatively, a new network adapter can be developed. The vhdl sample code in Listing 1 gives examples of what signals need to be used for the different transaction types for both masters and slaves. Also see Fig. 5 for a visual representation of the different transactions from the view of a master. The slave timing diagram would be similar but some of the signal directions are reversed.

```
-- master write to slave
if NA_not_ready = '0' then
   wr <= '1';
   read_request <= '0';
   targetid <= “0011”;
   addr <= x"0000000";
   wr_data   <= “00001111”;
end if;

-- master send read request to slave
if not_ready = '0' then
   read_request <= '1';
   wr <= '0';
   targetid <= "0001"; -- read from uart
   addr <= x"0000000";
end if;

-- master sending no read requests or writes
wr <= ‘0’;
read_request <= ‘0’;

-- master listens for read returns from slave
if read_return = '1' then
   read_return_register <= rd_data;
end if;

-- slave listening for incoming write
if wr = '1' then
   memory_array(addr_in) <= wr_data(7 downto
0);
end if;

-- slave listening for read request and
replying with read return
if read_request = '1' then
   rd_data <= memory_array(addr_in);
   read_return <= '1';
else
   read_return <= ‘0’;
end if;
```

Listing 1 - Network Interface 1

2.4 Routers

Routers have five sets of channels connected to it. As the mesh organization is used in the network, we can identify four directions: north, east, south, west and one additional channel going to the local network adapter. The channels are 49-bits wide and there is an input and an output channel for each adjacent router and the local network adapter. There are also input and output busy signals for each channel. The router is clocked with the global clock of the system and the reset signal is connected. The architecture consists of the boundary layer that holds and manages the input signals (see Fig. 6) and buffers and internal layer where the packets are routed to the suitable output channel or forwarded back to the buffers.

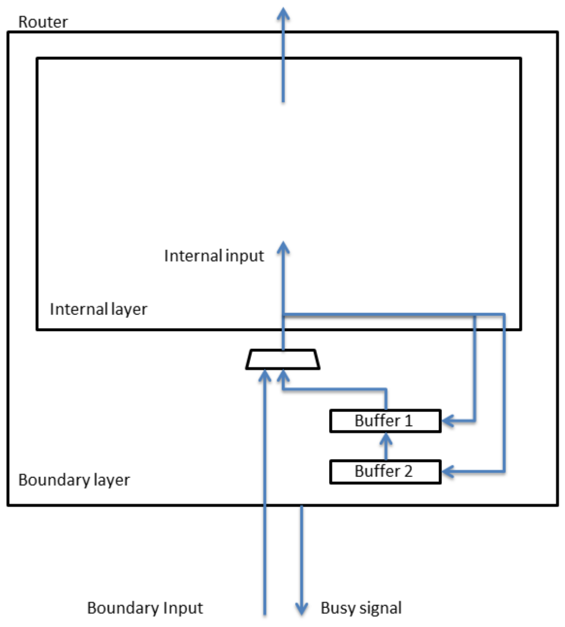


Fig. 6 Router layered structure with buffers

2.4.1 *Routing algorithm*

In order to move the packets around the network from source to destination, the routers look at the x-y counters of the incoming packets. Giving priority to the y-counter, the routers first check in what vertical direction they should forward the packet. If the negative flag attached to the y-counter is set to 0, then the packet is moved to the South. In the case where the negative flag is set to 1, the packet is moved to the North instead. As the packet is passed on to the next router, the y-counter is decremented by 1.

If the router receives a packet with a null y-counter, it then looks at the x-counter to determine in what horizontal direction the packet should move. This is done in a similar way to the y-counter. If the negative flag attached to the x-counter is set to 0, then the packet is moved to the East. In the case where the negative flag is set to 1, the packet is moved to the West instead.

If a router receives a packet in which both x and y counters are set to 0, then it indicates that the packet was destined for that router and that it has now reached its destination (and should be passed on to the local network adapter).

2.4.2 *Collisions*

Collisions occur when two or more input packets are trying to forward to the same output channel. To calculate this, the routing algorithm looks at each input separately in a loop, in the following order: (0) local, (1) north, (2) east, (3) south then (4) west (see Fig. 7). For each input, it first chooses a packet from either the buffer (if it is being used), the input channel or an empty packet if the channel is idle. If the packet is not empty, the desired direction is calculated and stored in a variable. Each input also has a corresponding collision flag. It first checks the busy input signal in the desired direction and if this is busy then the collision flag is set to ‘1’. It then loops through input 0 up to the current input (ignoring empty packets) and if any of these are also trying to forward to the same output then the collision flag is set to ‘1’. After all of this checking, the collision flag is forwarded to the next part of the algorithm to determine the next state of the output channels and input buffers, as described in 2.4.3.

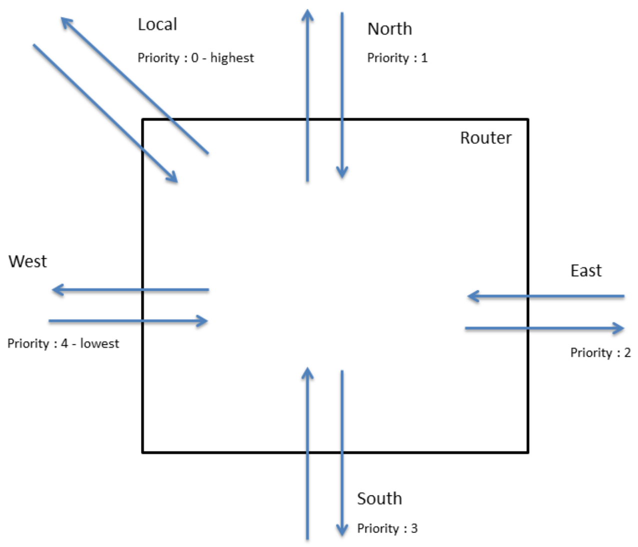


Fig. 7 Router channels coming from different directions

2.4.3 *Buffers*

If a collision flag is set (as described in 2.4.2), some packets must be stored in a buffer. Each router has a two-level buffer for every input channel. The implementation of a second level buffer is crucial in order to compensate for the delay it takes to inform the sending router about the busy state of the receiving router. Essentially, if the collision flag is set for an input, the corresponding input packet is stored in the first level buffer, the “First Level Buffer Used” flag is set and, simultaneously, a busy signal is sent back to the sender. However, this BUSY signal is only taken into account on the next output of the sender and a second packet might have already been sent to the receiver. If a second packet has indeed been sent in the meantime, it will be stored in the second level buffer and the “Second Level Buffer Used” flag is set.

2.4.4 Busy signals

There is a busy signal for every input buffer in a router and it is set to ‘1’ when the input buffer is being used. A busy signal is connected to the adjacent router or network adapter and tells it not to send any packets on this channel. If a packet is sent while a busy signal is asserted, it will be lost if both buffer levels are full.

3. INFRASTRUCTURE

3.1 Traffic counters and connection to MATLAB

From the outside world, it is difficult to see how the NoC is operating, particularly when there are many IP cores connected to the network. A connection has been made via a UART between the NoC running on the FPGA board and a laptop running a MATLAB program. The MATLAB program then displays a visual representation of the current traffic on the network which updates several times a second.

3.1.1 Traffic counters

Two sets of 128 8-bit registers have been added to the hardware. The first 48 registers correspond to the wires passing packets between routers and the other registers correspond to busy signals in the routers. On every clock cycle, every register in one set increments if the corresponding wires are currently being used or set to busy. After every 500 clock cycles it clears the other set of counters and starts counting in that set instead. At the same time, another process cycles through the registers and transmits the sum (from the set currently not incrementing) to the UART module (after waiting for the UART module to be not full). Note that the UART cycle will switch back and forth between the two sets several times during one loop since it won’t be able to transmit all 128 characters within 500 clock cycles. After cycling through the 128 registers it sends a terminator character.

These traffic counters and UART module add a significant amount of hardware and therefore should be commented out when implementing a real system.

3.1.2 MATLAB program

The MATLAB code is relatively simple since there are built in functions for setting up and reading from a UART. When a terminator character is received (should be every 128 characters), the data is sent to the drawing method. The drawing method allocates colours to specific counter ranges. Red wires indicate the wires have a lot of traffic, yellow indicates a small amount of traffic while white indicates no traffic. A similar colour scheme is being used for the busy signals except the ranges boundary values are smaller since the busy counters are generally much lower than the traffic counters. Once the figure has finished drawing, the UART buffer is first cleared (to receive the most recent data) and then waits for the next terminator character before repeating again.

An example of the MATLAB figure is shown in Fig. 8 and further explanation for specific examples is given in section 4.1.

4. EVALUATION AND TESTING

4.1 Example network

The following example shows an example network implementations by inserting *dummy processors*, memory modules and a switches input module into nodes in the network, as shown in Fig. 8. The interactions between the nodes are described below.

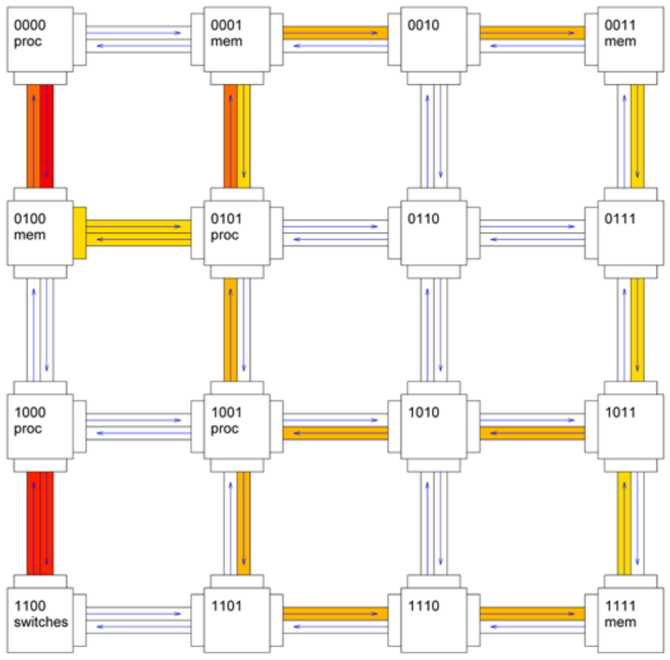


Fig. 8 Matlab figure - switches set to 1000

The *dummy processor* at node 0000 alternates between sending writes and read requests to the memory module at node 0100 every 4 clock cycles (and the memory module replies with read returns). This is why high traffic can be seen on the signals between node 0000 and 0100.

The *dummy processor* at node 0101 cycles every 10 clock cycles between a read request to 0100, a write to 0001, a read request to 0001 then another write to 0001.

The *dummy processor* at node 1001 cycles every 10 clock cycles between a read request to 0011, a write to 1111, a read request to 1111 then a write to 0011. It can be seen that the read request packets take a different route to the read return packets since the routing algorithm looks at the vertical components first (as described in 2.4.1).

The *dummy processor* at node 1000 is the only *dummy processor* whose behaviour can be changed during run time. It alternates every 2 clock cycles between a read request to the switches input module (linked to the physical switches *SW[7 downto 0]* on the DE2-70 board) at node 1100 and a write to the node with the ID given by the lower 4 bits of the read return. For example if the lower 4 switches are set to “0100”, the dummy processor will do a write to the memory module at node 0100 (see Fig. 9). Note that if the switches are set to a node that does not have a memory module then the packet will be ignored by the receiving node, however the packet still generates traffic through the network.

The smaller rectangles adjacent to the edges of the nodes indicate when the input buffers are being used (see 2.3.2). This occurs when the input packets coming in from that direction cannot proceed to the desired output channel due to that channel being busy (see 2.3.4) or an input channel with a higher priority wants to route to the same output channel (see 2.3.3).

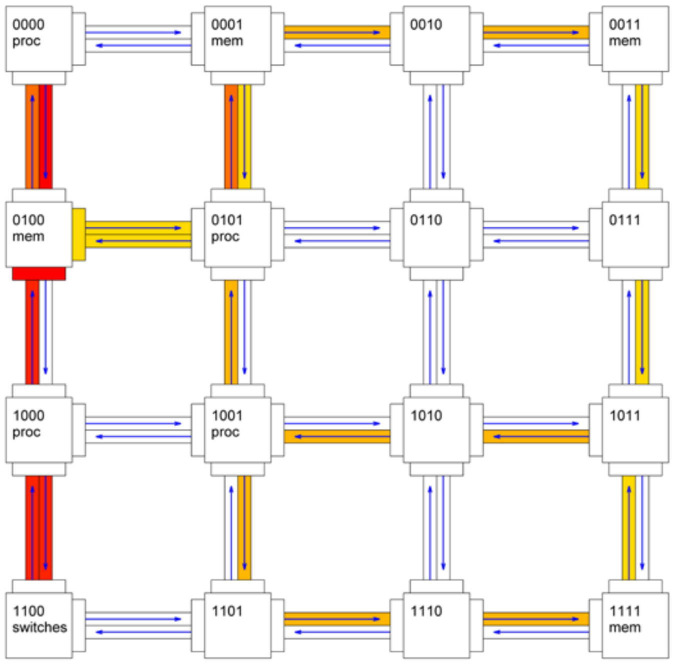


Fig. 9 Matlab figure - switches set to 0100

Fig. 9 shows the network traffic when the switches are set to node 0100. In some clock cycles there is a collision between the North, East and South packets all sending packets to the memory module at 0100. The busy signals show how the priority (see 2.3) is always given to the North input and therefore it is never waiting. However the East and South input packets must sometimes wait in the input buffers since their priorities are lower.

Fig. 10 shows the network traffic when the switches are set to node 0011. It can be seen that the busy signals are being raised at node 0100 south input (the south and local inputs are writing to north output) and the node 0001 west input (the west and south input are writing to the east output).

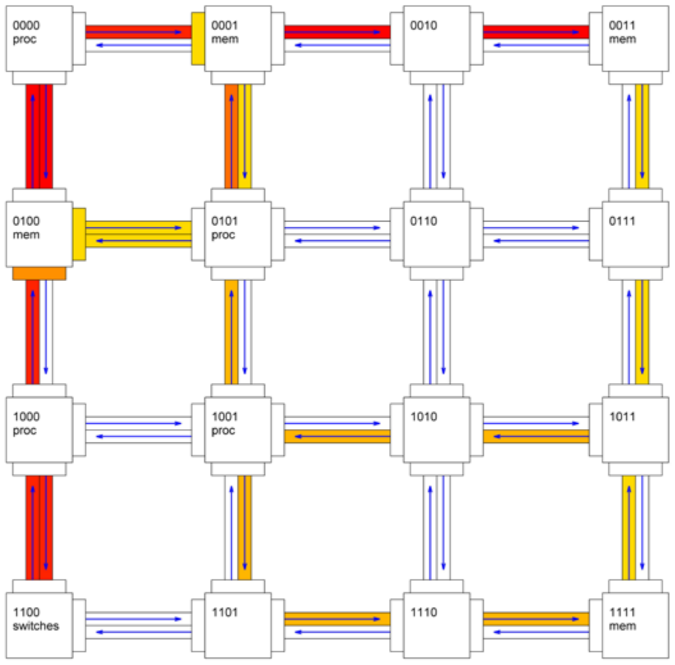


Fig. 10 Matlab figure - switches set to 0011

4.2 Numbers

It is natural that the number of logic units required for each module varies. This applies to both the number of combinational functions in the circuit and the dedicated logic registers. In the following table 1, statistics about the system were gathered from the compilation report.

Routers take up the most logic units and their size depends on their location within the network. This is because the circuit is optimized and unconnected wires and input buffers around the edges are erased from the system. For routers located on the corners of the grid, there are six channels for 3 directions and corresponding busy signals. For edge routers there are eight channels for 4 directions with corresponding busy signals. The network adapters are at least 20 times smaller than routers. This is because most of the complexity of routing was implemented in the routers, while the network adapters are there to build and decode packets and calculate the value of the routing counters.

It should be noted that the traffic counters and UART modules use a considerable amount of logic units and can be removed when implementing a real-world system. In a real-system, the size of the IP cores would use more logic units than the dummy processors and memory modules implemented in the example network.

Table 1 NoC Resource Utilization

Network-on-Chip elements	Total combinational functions	Dedicated logic registers
All nodes in NoC + statistics module	12 531	7 796
Master Network Adapter	7-12	15-22
Slave Network Adapter	13-26	19-32
Empty Corner Router	283	191
Empty Edge Router	334-336	226-227
Empty Internal Router	632-633	302
Corner Node with dummy Processor	401	263
Edge Node with dummy Processor	570	313
Internal Node with dummy Processor	868-873	410-412
Dummy Processor (used in this example)	62	52
Memory (used in this example)	51	73

5. Discussion

5.1 *Improvements*

There are several improvements that we can think of that may be necessary to implement if it being used for real world applications.

5.1.1 *Reduce number of registers/logic gates*

Our NoC has a large number of registers used for buffers (2*5*number of routers 49 bit registers = 7840 bits), and most of them are unused most of the time. We cannot think of many ways to reduce this number without significantly altering the routing algorithm. One possible idea is to have a smaller bank of registers that can be used by any router and have labels which indicate which router is currently using each register. This could allow a lower number of registers since it is unlikely that all routers will be using all 10 registers at any one time, but it is difficult to know how many are necessary.

5.1.2 *Larger data width*

We decided to use an 8 bit data bus which is converted to 8 bits of data per packet by the network adapters. We chose 8 bits because most of our testing was with a UART module, which only required 8 bits of data. However, realistic systems are likely to have a larger bus and our NoC should be able to adapt to that. A larger bus width would decrease the fraction of overheads in a packet. Unfortunately, this has all been hard-coded, it would be better if we used constants instead to be more flexible to changes.

5.1.3 *Buffers in network adapters*

All routers have 2 levels of buffers on every input in case a channel is busy. However, the network adapters do not have any buffer system and assume that any data is sends is being received correctly by the routers or IP core. Future versions of the code should take account the possibility that the network adapter’s outputs are not being received due to the network being busy for example.

5.1.4 *Variable network size*

We have fixed our mesh network to be 4 by 4 nodes. However, some applications may require more or less than 16 nodes for the network. Future versions of the NoC could allow more a more flexible network size. In some cases, this would need to alter the packet size due different sized x/y counters and the number of unique node ID’s.

5.1.5 *Out of order buffer queue*

Router buffers are filled based on the packet in the front of buffer not being able to proceed. However, packets coming in later could be able to proceed (if they have a different destination node) independent of the packet in the front of the queue. However if this was implemented, it may result in out of order packet delivery. It would also complicate the collision detection algorithm

5.1.6 *Buffer clearing delay*

For the same reason why we require 2 buffer levels, there is a 1 clock cycle delay between when the buffer clears and when the next packet can be received. We cannot think of any ways to avoid this.

5.1.7 Bidirectional network adapters

All of our network adapters are either master network adapters or slave network adapters. This is suitable for most applications since most IP cores can be considered either a master or a slave device. However, some applications may require IP cores that have the behaviour of both a master or slave node. For this reason, it may be necessary to create another network adapter type that allows this behaviour.

5.1.8 Turn off modules

Most of the time, most parts of the NoC will be idle. However, they will still use up power and receive a clock signal. To be more power efficient, it would be better if some modules could be turned on and off based on when they are being used. We are not sure if this is possible on an FPGA.

5.1.9 Torus topology

A relatively simple extension to our mesh topology would be to convert it to a circular-mesh or torus network. This would involve connecting all of the unused West channels in column 00 with the East channels in column 11 and similarly with the unused North and South channels. This would be simple to implement since there would be no changes required in the routers. There would be small changes required in the counter initialisations in the network adapters but the increase in logic units would be negligible. Finally, the channels would have to be connected by wires. The advantages would be a reduced (or equal) number of hops for all packets. It could potentially decrease traffic congestion since there are more paths available. However, all routers would be “internal” rather than “corner” or “edge” routers, as described in 4.2, which would increase the number of logic units.

5.2 Alternatives

There has been a growing amount of research on Network-on-a-Chip design and the following sections outline some alternative network design decisions that we could have chosen instead.

5.2.1 Multiple flits per packet

We chose to keep the design simple by having one flit per packet. However, it is also possible to break packets up into multiple flits. This would allow larger packet sizes and/or a reduced channel width. However, multiple flit packets would also greatly increase the complexity of the routing algorithms and would now have more of a problem with deadlocks. There would also be more work for the network adapters because the packet would be created over several clock cycles.

5.2.2 Strings of packets

This idea is similar to the “multiple flits per packet” idea, except that it is multiple packets with a common source and destination. It is quite common for an IP core to send multiple consecutive reads or writes to a memory module or to a cache for example. In our NoC, if this behaviour were to occur, each packet would have to be individually addressed and routed. However, it could be possible for a master to send an instruction similar to “the next x write packets should be received by node y” or have a start and end of string packets. This could potentially decrease the packet overheard for strings of data, however it would also complicate the design.

5.2.3 Different network topologies

We chose a simple 2D mesh topology since this is the easiest to visualise. However, there are many other possibilities of network topologies that could be used such as a 3D mesh, star network or torus (see 5.1.9). Each of these has its own advantages and disadvantages.

5.2.4 Routing algorithm

Our NoC has taken the simple approach by giving each packet a fixed path. However it could be possible for alternate paths to be given by the routers based on the traffic on the channels. This could decrease the packet routing time but would also increase the amount of logic required in the routers. A more extreme case of this is called “hot potato” routing where the router must never hold a packet and always pass it on to another router. It must pass a packet on even if the only direction available takes the packet further away from the destination. This could mean the routers no longer need buffers, however it could also mean that packets are being passed around the network and never reach the destination. This would also increase the amount of logic required in the routers.

5.2.5 Routing priorities

We chose to use statically allocated priorities given to each input of a network adapter. This decreases the design complexity and in most cases greatly reduces the required routing logic. However, it could be possible to have priorities that change dynamically, such as the input priorities changing in a round robin fashion. This could potentially prevent a particular packet from being blocked for an extended period of time due to a large amount of traffic from other directions with a higher priority. Another alternative would be to have an IP core give a priority to each packet so that more important packets can be routed through the network faster.

5.2.6. Address space division

Our NoC has a fixed 32 bit address space where the first 4 bits are allocated to node IDs and the other 28 bits are for local addressing. This means that every node is allocated 2^28 address locations even when the IP core at the node only requires a single or small number of address locations. At the same time, this can be a restriction on the size of memory modules for example where a large number of addresses are required. In most computer systems, the address space is not divided evenly between every module; instead it is based on the number of locations required by each module. In our NoC, the network adapters must have knowledge of how the address space has been divided so that it can calculate the destination node and initialise the x/y counters correctly. If the address space was not divided evenly, some kind of look-up table would be required for the network adapters to calculate the destination nodes.

5.3 Future Work

The following sections outline some future work that could be undertaken if the project was continued.

5.3.1 Deadlocks

A common problem in NoC design is to be able to ensure a NoC is not affected by deadlocks or if a deadlock occurs, overcoming it. Our design and the results from stress testing indicate that deadlocks cannot be encountered in our NoC, possibly due to the fact that some paths are not possible (eg turning from horizontal to vertical) and due to having 1 flit per packet. However, this has not been formally proven and this would be necessary if implementing in a real-world system.

5.3.2 Local clocks

This is not so much a problem in FPGA's with a low clock speed; however in modern circuits there has been a growing need to use local clocks or even using asynchronous logic. This is due to the possibility of having a clock skew and clock jitter between different parts of the circuit. In the case of our design, the NoC has been implemented with one common clock shared between all modules. Local clocks or asynchronous circuitry could be considered in future designs.

5.3.3 Standard interface

We have designed a simple bus connection between IP cores, network adapters and memory modules. If the NoC was implemented in a real system, the modules would have to be able to communicate using our bus protocols. Alternatively, more variations of network adapters could be developed to allow standard bus protocols (such as CAN) to be used instead. This would allow the NoC to be used in a wider range of applications.

6. CONCLUSION

In this project, we have successfully designed, implemented and tested a Network-on-Chip on an FPGA board. There were many ways we could have approached this design, however we decided to keep it as simple as possible. The choice of one flit per packet for a system such as ours proved to simplify the design and avoid complications such as deadlocking. The choice of using deterministic paths and fixed priority routing also helped reduce the complexity of the system. It is possible to see how these algorithms and designs can be applied to other larger Network-on-Chips. We have outlined some ideas for improvements and future work to the project. This includes developing a more scalable and universal design to allow it to be used in real-world applications.

Please have a look at our open-source project which can be found at: *github.com/mattbirman/Network-on-Chip-in-VHDL*

REFERENCES

Jens Sparsø, "Networks on Chips - basics", *02211 Advanced Computer Architecture lecture slides*, 2012.

Jens Sparsø, "Networks on Chips - examples", *02211 Advanced Computer Architecture lecture slides*, 2012.